Chapter 1

Introduction to Software Engineering/Development

In this chapter, we will define what we mean by "software engineering", briefly touch on the roots of software engineering and how it came about, and compare it to computer science and other engineering disciplines. Software Engineering refers to applying engineering principles to software development. Software engineering and software development, although are sometimes used as synonyms, they do not mean the same.

1.1 What is software?

Before we can understand the meaning of "software engineering" or "software development", we must first understand the meaning of "software".

Software is both a solution and a problem Generally speaking, software is built as a solution to a problem. Of all the types of software systems that exist, most are built as a solution to a problem. Some examples are:

文字处理器

1. Word processors: The first word processors were built because composition, editing, and formatting of documents was difficult using typewriters.

供应链物流系统

2. Supply-chain logistics systems: A business may require some custom-built software to keep track of supply-chain logistics, because the current manual system they employ is inefficient and error prone.

3. Games: Games are typically built to provide entertainment, and thus relieve us of the problem of boredom.

As such, software engineering is a problem-solving activity. To solve problems, most engineering disciplines have evolved the following steps to building their systems:

1. identify the problem;
2. analyse the problem;
3. derive solutions to the problem;
4. choose the most appropriate solution to the problem; and
5. realise that solution.

Despite being a solution, a piece of software is also a problem in its own right. The five steps above are problems that must be tackled in themselves as part of building a software system.

One could say that a software system solves a business problem, but creates an engineering problem. Perhaps one could claim that for a software system to be successful, it would need to solve a harder problem than its own construction.

There are different types of software systems, and these different types of systems require different types of solutions. A software system may solve a problem for a specific customer according the their requirement; for example, a customer who wants supply-chain tracking software; or may be solve a problem for a general market; for example, a word processor or web browser. The tools and techniques used to solve these systems may be different. For a custom product, one may be able to elicit the requirements by reading some information and performing interviews with the customer. For a general market product, one may want to do market studies to gather information about what users would like. Even the technical details may be affected. For a custom-made product, you may know the exact details of the hardware and platform on which the product will run; or it may even be developed for a piece of unique piece of hardware, of which only one instance will ever be built. For a general market product, one would typically want to make the product as portable as possible. In the former case, you may use a low-level programming language to allow control of the hardware, whereas in the latter case, you may want to use cross-platform programming language.

This subject is about the different approaches to solving problems using software. Software systems are not simply UML + Java + JUnit. In most cases, such an approach will not be suitable, and in some, not possible at all.

Software System Software in isolation is worthless. To be useful, software must interact with hardware devices that provide it with input and output, whether is be keyboards and PC monitors, or sensors and actuators in a rail system. As such, software is developed as part of a larger system.

A *system* is a collection of entities that are inter-connected. Thus, when we speak of a *software system*, we are talking about a set of programs that are inter-connected or related. When we speak of a *system* in general, we are talking about a set of programs, hardware components, networks and other devices that are inter-connected and work together to achieve a common goal.

Many of the requirements placed upon software often originate from the constraints imposed by hardware, communications networks, and other software. The design of a software system typically abstracts away from the details of the networks, operating systems and hardware. Instead, designs rely on *application programmer interfaces* (APIs) that allow developers to interact with hardware platforms without worrying about details of that interaction.

An API, however, does not completely isolate us from networks and hardware. For example, it takes time for messages to be sent along a network and it takes time to acquire resources such as printers and data bases.

Sensors, such as temperature sensors, motion detectors, RFID sensors, and cameras are other examples of devices that will not only influence your software design, but are often key parts of your solution to a specific engineering problem. In practice systems involving sensors require a complete *systems approach to design*.

A systems approach is required for the design many programs, especially if they combine hardware, software and networks into a single entity. For you this means much more than having to just think about delivering functions. It means having to think about:

- the physical layout of computers, network cables and the positioning of sensors, and other hardware;
- the reliability of components and how this affects the reliability of the system as a whole;
- the performance of components and how it affects the system as a whole; and
- other factors that will influence the behaviour of the system as a whole and how the system meets its goals.

When we are writing a piece of software we are actually creating a system. Every piece of software we write will eventually run on a *platform* – a computer executing a specific operating system possibly (probably) connected to a network and devices such as printers, scanners, PDAs or sensors. In designing and testing a system, you will need to take into account the latencies due to networks, communication with external devices such as sensors and interaction with other systems.

1.2 Defining software engineering

The IEEE 610.12-1990 standard [3] defines software engineering as follows:

1. The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
2. The study of approaches as in (1).

Thus, software engineering is both the application of engineering principles to the development and maintenance of software, as well as the study of how this can be done.

The Software Engineering Body of Knowledge (SWEBOK) Guide [1], completed in 2014, identifies ten *knowledge areas* that comprise software engineering:

1. software requirements;
2. software design;
3. software construction;
4. software testing;
5. software maintenance;
6. *software configuration management*;
7. *software engineering management*;
8. *software engineering process*;
9. *software engineering tools and methods*; and
10. *software quality*.

More recently, SWEBOK V3 [4], adds the following five knowledge areas to SWEBOK:

1. software engineering professional practice;
2. software engineering economics;
3. computing foundations;
4. mathematical foundations; and
5. engineering foundations.

Students will have already studied some of these topics, such as software design. The knowledge areas in *italics* are those that will be the main focus of this subject, although we will touch on others as well.

As a discipline, there are many different aspects to software engineering. In a typical software engineering project, there will be many different areas of application, such as:

1. computer science theory;
2. processes;
3. project management;
4. methods and techniques for planning and measurement; and
5. experience.

Each of these play an important role in most projects, and without them, many projects would fail.

1.3 Computer science vs. software engineering

The critical difference between computer science and software engineering is evident from the names. Computer science is a science, while software engineering is an engineering method. More accurately, computer science is the theoretical foundation on which software engineering is built. A software engineer applies the results from computer science as problem-solving tools in their projects. This relationship is similar to that of chemistry to chemical engineering, or physics to electrical engineering.

However, the inherent difference between computer science and software engineering is *complexity*. Generally, the complexity is a result of the size of the system, but other factors are also an influence. Put simply, computer science is concerned with solving small-scale problems regarding certain computational problems, while software engineering is concerned with how to build and manage large-scale systems. The complexity of large-scale systems is generally so great that it is not possible for one person to understand the system, so it is necessary for teams of people to coordinate with each other for a software project to succeed.

Example 1.1. *The Linux operating system kernel*

The first Linux operating system kernel, which was designed and implemented by Linus Torvalds entirely on his own, consisted of just over 10 thousand lines of code. Linux 4.2, released in June 2015, consists of almost 19.5 million lines of code, and has been developed by thousands of people over almost two decades. The Linux kernel could simply not have been produced without following sound software engineering principles, due to its inherent complexity. Software engineering is about dealing with this complexity using art, science, and engineering. There are several planning, problem solving, monitoring, and controlling disciplines involved in the development now. For example, the Linux kernel itself has several different branches, with each branch having several levels of stability. Such a task requires a configuration management process that must be strictly adhered to. Such a process is part of the larger software engineering method that is followed by Torvalds and his army of developers.

As a result of the large-scale nature of many software systems, software engineers typically take a *top-down* approach to building software. That is, they gather a good understanding of the problem at a high-level of abstraction, analyse this problem, and then break this problem into smaller problems that are easier to understand and solve. From this, they then use computer science as the tool to solve these smaller problems, and then synthesis the entire solution from these smaller building blocks; a *bottom-up* approach.

1.4 Software engineering vs. other forms of engineering

Software engineering is closely related to other forms of engineering. Some of the similarities include:

- All forms of engineering are concerned with building reliable products that solve some problem.
- The use of science, maths, and empirical knowledge to assure quality of the products, and to reduce the cost of building and maintaining products.
- The use of large teams to build large-scale products.
- Most engineering disciplines suffer from the problem of changing requirements. For example, civil engineers will have customers change their preferences during the design of a building, electrical engineers will change their mind during the design of a portable device, etc. Similarly, software engineers must account for the changing requirements of their customers.

Despite these similarities, there are many differences between software engineering and most other forms of engineering. Some of these factors include:

Age: Software engineering is an immature discipline, spanning roughly the last 50 years. Civil engineering, on the other hand, is thousands of years old. Software engineers have taken advantage of the wisdom and experience gained in other engineering disciplines to apply them to the software.

Cost: The cost of construction materials and labour (other than engineering labour) is significantly higher in other engineering disciplines, relative to the costs of a a compiler and a PC. As such, the percentage of engineering in a software system is notably higher than in other engineering disciplines, so delays and mistakes have a larger impact.

Flexibility: Software is generally quite flexible. This is related to the previous point on cost. It is more straightforward to change something about a software system after it is built than for many other types of engineered products. For example, you would never ask a civil engineer to move a bridge 50 metres downstream one month before opening. However, software engineers are routinely asked to change major parts of a project just prior to release. Only with proper control can this be achieved.

Innovation: Typically, software is more innovative than other engineered products. Due to the relative ease and low cost of replicating a piece of software, software engineers are often solving a problem that has never been solved before. As a result, software engineers often have to use more innovative solutions.

Domain specificity: While all engineering projects have constraints on the product, such as their environment, software engineering projects are almost always domain specific, and often this requires the software engineers to understand to most important parts of the domain. For example, when constructing an office building, it is often largely irrelevant what the business of occupants of the building will be. However, for software, the domain of application is highly relevant.

Complexity: The level of complexity of a software application is generally much higher than in other forms of engineering. Large software systems contain a number of variables with many interactions between them that are difficult to understand.

Reliability: Software systems are typically more unreliable than other engineered products. This is largely due to the factors already discussed: products are more complex; the field is less mature, so the tools and techniques cannot cope with this complexity; more innovative solutions are required, so there is less scope to use tried and tested techniques; software engineers cannot often obtain a good enough understand of the domain of the software; and the cost of replicating and re-deploying software is low enough that unreliable systems can be replaced more easily. Furthermore, people tolerate unreliable software systems more than other engineered products. An unreliable bridge, electrical product, chemical, etc., are likely to cause harm, whereas the average desktop application will not. We do find, however, that safety-critical software systems are engineered more carefully and rigorously than standard desktop application for this very reason, and are more reliable.

1.5 Why do we need software engineering?

A valid question may be: "Is it possible to have a successful software project without software engineering?". The answer, quite simply, is "Yes, it is possible". However, a more accurate answer is: "Yes, it is possible, but it is unlikely". It is certainly possible for a group of genius programmers to get together and hack out a quality product, but this would be largely due to luck and individual brilliance. Unfortunately, there are not enough charmed geniuses in the world to support even a fraction of the requests of those wanting quality software, so we use software engineering to provide *systematically* and *repeatedly* build quality software.

The development of software engineering as a field came about from necessity. People building software found that they encountered many problems during construction, and so produced techniques to mitigate these. They found that applying many of these systematically results in repeatedly being able to build quality software, so they share these with the rest of the community. There are many different tools and techniques for many different types of systems, but software engineers will only create and adopt these tools and techniques if they believe that they help to solve some problem.

The reason the discipline of software engineering is becoming more important is because software is become more and more complex. As the power of computing increases, and the size of problems that can be solved in a reasonable time becomes larger, the ability to conceive and develop these solutions becomes more difficult.

Software engineering helps to manage this complexity using four broad disciplines: *monitoring*, *controlling*, *analysis*, and *synthesis*. This subject will cover each of these disciplines in detail.

What accounts for most of the failures in software projects? The answer to this depends on which era you are talking about. Over time, software engineers have become better at some activities more than others. Recent empirical evidence identifies the following reasons for failure:

Misunderstanding of requirements	53%
Design failures	22%
Project management failures	13%
Other	12%

The "other" entry here includes failure in team work, incorrect choices of technology, insufficient testing, insufficient configuration management. Therefore, what many people may consider as the typical software "bug" — that is, a programmer writing code that behaves differently to what they intended — account for a small amount of software failures of released systems. Despite much of the careful planning and hard work that goes into many requirements documents, requirements account for more than half of all failures.

1.6 The history of software engineering

Software engineering as a discipline was born at the 1968 NATO Software Engineering Conference, which was called together to address the *software crisis* at the time. As a result of the rapid increase in computing power, the complexity and size of the problems that could be solved in a reasonable amount of time exploded, and computer programmers discovered their ability to conceive and manage solutions diminished.

Respected computer scientist Edsger Dijkstra outlined the problem with the following (now famous) quote from his 1972 Turing Award Lecture [2]:

> "To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."

As a result of the software crisis, programmers found that their projects continued to run over time and over budget, and that their software products were of low quality, failed to meet user requirements, and were difficult to maintain.

50 years later, and we still experience these same problems with software, so some may claim that the discipline of software engineering has not provided any solutions.

However, what is undebatable is that the complexity of software over the last 50 years has increased several orders of magnitude, and that the type of problems solved by software today would have been a pipedream 50 years ago.

Example 1.2. *Great software failures*

There are many examples of software failure that have cost millions of dollars and, in extreme cases, people's lives. Some well known examples include:

- Ariane 5 Flight 501. In 1996, the first of the European Ariane 5 expandable launch system rockets exploded 37 seconds after take off. The rocket engaged an automatic self-destruct system after it veered off course and extreme aerodynamic forces caused parts of the hardware to fail. The failure to correctly follow its programmed course was due to the incorrect use of a module from the Ariane 4 systems. A 64-bit floating point number was being converted into a 16-bit integer number, but the value of the floating point was large enough to cause an overflow. In the Ariane 4 systems, the flight path of the rockets were programmed such that this was never a problem. However, the altered flight path of the Ariane 5 systems was of a considerably different range for this to cause the error. Formal investigations revealed that the error handling to this conversion had been disabled to increase efficiency. Estimated cost: US $370 million.

- The Mars Climate Orbiter. In 1999, the Mars Climate Orbiter spacecraft approached Mars as part of its missions to monitor and study the weather and climate conditions on Mars. As it approached the planet, it entered orbit at a lower altitude than intended by its designers, and was destroyed by the atmospheric pressures. The cause of this error was due to the incorrect metric usage: the controlling software on Earth was using the metric system, while the spacecraft was using the imperial system. As with the Ariane 5 failure, the software had been adapted from a previous craft without being sufficiently verified. Estimated cost: US $327.6 million.
- The Myki ticketing system. In 2005, the Victorian government awarded a AU $500 million dollar contract to produce an electronic train, tram, and bus ticketing system, called Myki. The new system was planned to go live by the end of 2007. However, the system was not rolled out until December 2009, even this has been plagued with problems, with the ticketing authority halting the distribution of new cards. If media sources are to be believed, the total cost so far has ballooned to AU $1.3 billion. The cause of the delays appears to be related to the fare calculations based on the Global Positioning System (GPS). With several cities in Australia (Brisbane, Sydney, Perth, and Adelaide) and around the world (London, Hong Kong, and Singapore) having already rolled out electronic systems with GPS-based fare calculation at a fraction of the cost, the failure can be accounted as a managerial (or political) failure (by not buying an existing "off-the-shelf" system) as much as a technical failure. Estimated cost: AU $1.3 billion, AU $630 million over the initial estimate.

References

[1] A. Abran, P. Bourque, R. Dupuis, and J.W. Moore. *Guide to the Software Engineering Body of Knowledge – (SWEBOK)*. IEEE Computer Society, 2004.

[2] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[3] IEEE. IEEE standard glossary of software engineering terminology. IEEE Std 610.12-1990, 1990. IEEE Computer Society.

[4] IEEE Computer Society, Pierre Bourque, and Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edition, 2014.